Contents

About these reading notes

These are my own personal reading notes that I took (me, Franklin) as I read the textbook. I'm providing these to you as an additional resource for you to use while you're reading chapters from the textbook. These notes do not stand alone on their own — you might be able to get the *idea* of a chapter while reading these, but you're definitely not going to **get** the chapter by reading these alone.

These notes are inconsistently all of the following:

- Me summarizing parts of the text.
- Me commenting on parts of the text.
- Me asking questions to myself about the text.
    - … and *sometimes* answering those questions.

The way that I would expect you to read or use these notes is to effectively permit me to be your inner monologue while you're reading the textbook. As you're reading chapters and sections within chapters, you can take a look at what I've written here to get an idea of how I'm thinking about this content.

Chapter 15: Mechanism address translation

- **Note**: we skipped chapter 14; it's an overview of the use of the memory API (`malloc` and `free`). If you feel like you want a refresher on that, feel free to read this chapter. The one important nugget from chapter 14 is that `malloc` and `free` are **explicitly not** part of the operating system, but instead are part of the standard C library.
- At the start here the authors are writing about letting software run directly on the hardware with the aim of performance. What other options even are there? How else *could* software run, if it's not running directly on the CPU? (hint: think about Java, think about *emulators*)
- Efficiency and flexibility are two main goals here. Efficiency is specifically related to "run it on the hardware", and flexibility is "with OS support".
- Before getting too far (e.g., while you're still looking at pg 2): even thinking back to the basic idea that different processes get different spots in memory like in figure 13.2, based **entirely** on what you know about how processes get switched, try to theorize what the OS might *need* and what it might *do* to help hardware make sure that we can make these processes believe that they "start at 0".

15.1 Assumptions

- "Go ahead, laugh all you want; pretty soon it will be the OS laughing at you" `D:`
- Do you think that the assumptions listed here are realistic? (e.g., contiguous block of memory, and that address space is less than the size of physical memory)

15.2 An example

- `objdump` and `otool` are pretty cool programs, but you should also check out https://godbolt.org/, it is **amazing**.
- The sequence of what happens at the bottom of pg 3 starts with "Fetch instruction at address 128". Remember how "instructions are stored in memory"? Yeah. That means that to *run* an instruction, you have to *load* the instruction from memory. Neat.
- In terms of transparency and figure 15.2, again, try to stop here and think about what an OS might need to do to help a program believe that its address space starts at 0.

15.3 Dynamic (Hardware-based) relocation

- Virtual address: Remember that some addresses are set in a program **when the program is compiled**, and that virtual addresses start at 0. Other addresses are *effectively* generated at runtime (e.g., dynamic memory allocation).
- What do you think the OS is doing to help this all out? The hardware seems to have the main responsibility in dynamic relocation.

Example translations

- The authors describe the result of attempting to access a memory location out of bounds is a "fault, causing an exception to be raised". What do you think is the role of the operating system in this?

15.4 Hardware support: summary

- Do you think the OS itself needs to use the base and bounds registers for its own execution? Or are they *only* for user programs?
- Do you think a user program has the ability to read from and write to the base and bounds registers? Why or why not?

15.5 Operating system issues

- The authors describe a “free list”, what do you think is the difference between tracking “free” space and “used” space? We sort of tracked both with file system bitmaps.
- The authors mention that variable sized address spaces is a problem for future chapters. What kinds of problems do you think there would be with variable sized address spaces? (hint: it’s *also* a problem with file systems).
- Ugh, they’re referring to page 5 on page 10, it’s not easy to flip around with a PDF like this.
- “the OS must do some work when a process is terminated …, reclaiming all of its memory for use in other processes or the OS.” – is this different from `free`?
- In the middle of pg 10 is the answer to some above questions (when/how does the OS deal with base/bounds registers?)
- Figure 15.5 does a really nice job of showing 1) when things happen as the OS and hardware are initialized, and 2) which of hardware or the OS is responsible for different parts of this initialization.
    - Figure 15.6 is an *even more detailed* version of this diagram.

15.6 Summary

- This is all well and good, but, uh, surprise! Base and bounds is **not** what we’re doing with modern hardware and operating systems. The authors have a few good reasons why on pg 13. Can you think of any others?